CHAPTER 6

Introduction to Shell Programming

case Dominion Consulting needs a program to maintain its phone records file, which contains the following information about each employee: telephone number, name, department, job title, and date of hire. This program should let users add, change, delete, locate, and display specific employee information. You can meet these needs by creating a UNIX shell program. In this section, you learn the basic tools used to create UNIX shell programs. In Lesson B you create shell programs that let users view and add records to the phone database.

LESSON A

objectives

In this lesson you will:

- **Understand the program development cycle using a high-level computer language and UNIX shell scripts**
- **Compare the shells to determine the best choice for programming**
- **Learn about shell variables, operators, and wildcard characters**
- **Write simple shell scripts to illustrate programming logic**

Using the UNIX Shell as a Programming Language

Previewing the Application

The corp_phones file, which you create in this chapter, contains records with fields delimited by colons. Here is a sample record from the file:

```
219-432-4587:Mitchell:Barbara:C:4541:Admin Asst:12-14-1995
```

In Chapters 4 and 5 you learned several commands, such as grep, cut, paste, and awk, which you can use to manipulate, extract, and format information stored in files. Although these commands are powerful, they can be difficult for non-technical users. In addition, you must often use these commands together in long sequences to achieve the results you want. Repeatedly executing these command sequences can be cumbersome, even for the experienced technical user. UNIX shell scripts eliminate both of these problems. You can write shell scripts that present user-friendly screens and automatically issue commands like grep and awk to extract, format, and display information. This gives the non-technical user access to powerful features of UNIX. Shell scripts also save you time by automating long command sequences that you must perform often.

The shell script application you develop in this chapter and in Chapter 7 presents a menu of operations the user may choose from. Among other tasks, these operations automate the process of searching for, formatting, and displaying the phone number records in corp_phones. The application also provides data entry screens to ease the task of adding new records to the file.

To accomplish this chapter's case project, you will learn about these programming features of the UNIX shell:

- *Shell variables*: Your scripts often need to keep values in memory for later use. Shell variables are symbolic names that can access values stored in memory.
- *Operators*: Shell scripts support many operators, including those for performing mathematical operations.
- *Logic structures*: Shell scripts support sequential logic (for performing a series of commands), decision logic (for branching from one point in a script to another), looping logic (for repeating a command several times), and case logic (for choosing an action from several possible alternatives).

In addition, you will learn special commands for formatting screen output and positioning the cursor.

Before you begin writing shell scripts, you should understand the program development cycle.

The Program Development Cycle

The process of developing an application is known as the **program development cycle**. The steps involved in the cycle are the same, whether you are writing shell scripts or high-level language programs.

The process begins by creating program specifications—the requirements the application must meet. The specifications determine what data the application takes as input, the processes that must be performed on the data, and the correct output.

After you determine the specifications, the design process begins. During this process, programmers create file formats, screen layouts, and logical algorithms (procedures) that the program will use. Programmers use a variety of tools to design complex applications. In the next chapter, you will study flowcharts and pseudocode, two important program design tools.

After the design process is complete, programmers begin writing the actual code, which they must then test and debug. When they find errors, they must correct them and begin the testing process again. This procedure continues until the application performs satisfactorily.

Figure 6-1 illustrates the program development cycle.

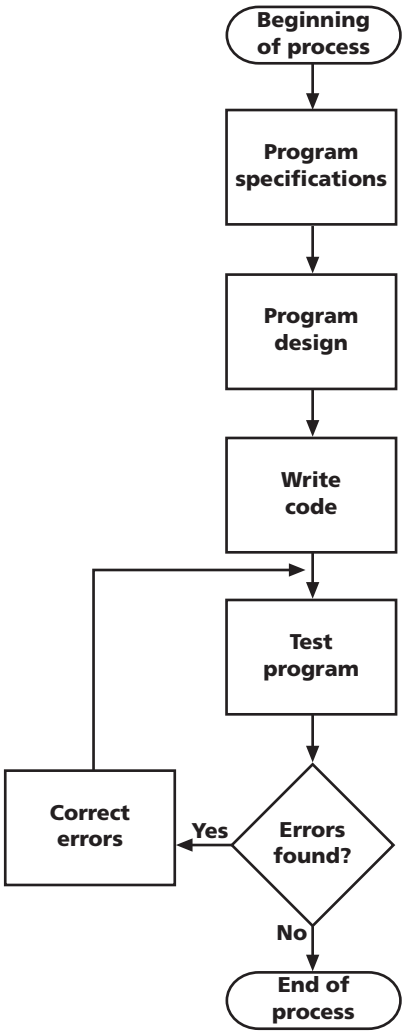

Figure 6-1: Program development cycle

Using High-Level Languages

Computer programs are instructions often written using a high-level language like COBOL, C, or C++. A **high-level language** is a computer language that uses English-like expressions. For example, the following COBOL statement instructs the computer to add 1 to the variable COUNTER:

```
ADD 1 TO COUNTER.
```

Here is a similar statement, written in C++:

```
counter = counter + 1;
```

A program's high-level language statements are stored in a file called the **source file**. This is the file that the programmer creates with an editor such as vi or Emacs. The source file cannot execute, however, because the computer can only process instructions written in low-level **machine language**. As you recall from Chapter 3, machine-language instructions are cryptic codes expressed in binary numbers. Therefore, the high-level source file must be converted into a low-level machine language file, as described next.

The source file is converted into an executable machine-language file by a program called a **compiler**. The compiler reads the lines of code that the programmer wrote in the source file and converts them to the appropriate machine language instructions. For example, the Linux C++ compiler is named g++. The following command illustrates how it is executed:

```
g++ -o my_exe my_source.cpp
```

In this sample command, the –o option followed by my_exe instructs the compiler to create an executable file, my_exe. The source file is my_source.cpp. The command causes the compiler to translate the C++ program my_source.cpp into an executable machine-language program, which is stored in the file my_exe.

If a source file contains **syntax errors** (grammatical mistakes in language usage), it cannot be converted into an executable file. The compiler locates and reports any syntax errors, which the programmer must correct.

> **Note: After compiling, the executable program may still contain fatal run errors or logic errors. Fatal run errors cause the program to abort due to an invalid memory location, for example. Logic errors cause the program to produce invalid results because of problems such as flawed mathematical statements.**

Another way to accomplish programming tasks is to develop UNIX shell scripts.

Using UNIX Shell Scripts

Introduced in Chapter 4, UNIX **shell scripts** are text files that contain sequences of UNIX commands. Like high-level source files, a programmer creates shell scripts with a text editor. Unlike high-level language programs, however, shell scripts do not have to be converted into machine language by a compiler. This is because the UNIX shell acts as an **interpreter** when reading script files. As an interpreter reads the statements in a program file, it immediately translates them into executable instructions, and causes them to run. No executable file is produced because the interpreter translates and executes the program statements in one step. If a syntax error is encountered, the program halts.

After you create a shell script, you simply tell the operating system that the file is a program that can be executed. This is accomplished by using the **chmod** ("change mode") command to change the file's mode. The mode determines how the file may be used. Recall that modes are denoted by single-letter codes, the most common being "r" (read), "w" (write), and "x" (execute). Further, the chmod command tells the computer who is allowed to use the file: the owner (u), the group (g), or all other users (o). For a complete description of the chmod command, see Appendix B, "Syntax Guide to UNIX Commands."

As you recall from Chapter 4, you can change the mode of a file so that UNIX recognizes it as an executable program that everyone (user, group, and others) can use. In the following example, the user is the owner of the file.

```
$ chmod ugo+x filename      <Enter>
```

By typing the filename after the system prompt, you can now run the shell program.

However, before any program can be run, it must be retrieved from a path identified in the **PATH variable**, which provides a list of directory locations where UNIX looks to find the executable programs. (See the section, "Variables," later in this chapter.) If the program resides in your current directory, which is not in the PATH variable, you can precede the program name with a dot slash (./) to tell UNIX to look in the current directory to find the program, as follows:

```
$ ./filename      <Enter>
```

Shell programs run less quickly than do compiled programs, because the shell must interpret each UNIX command inside the executable script file before it is executed. However, it generally takes less time to create a shell program than one written using a high-level language, because the process of invoking a compiler and producing an executable file is eliminated. Because it is usually faster and easier to write a program such as a shell script than a compiled language, many programmers prefer scripts.

Prototyping an Application

A **prototype** is a running model of your application, which lets you review the final results before committing to the design. Using a shell program to create a prototype is often the quickest and most efficient method, because prototyping logic and design capabilities reside within UNIX.

After the working prototype is approved, the program can be rewritten to run faster using a compiled language such as C++. If the shell program performs well, however, you may not need to convert it to a compiled version.

Now that you better understand the program development cycle, you realize that it will be more efficient to maintain Dominion Consulting's phone records using a shell script. Your next step is to select the programming shell that will be most efficient for your application.

The Programming Shell

All Linux versions use the Bash shell (Bourne Again Shell) as the default shell. Table 6-1 lists the three shells that come with most Linux distributions, their derivations, and distinguishing features.

Shell Name	Original Shell It Is Derived From	Description
Bash	Bourne and Korn shells	Offers strong programming language features like shell variables, control structures, and logic/math expressions. Combines the best features of the other shells.
tsch	C Shell	Conforms to a programming language format. Shell expressions use operators like those found in the C programming language.
zsh	Korn shell	Reflects the Bash shell in many respects but also has C programming-like syntax. Useful if you are familiar with older Korn shell scripts.

Table 6-1: Linux shells

The Bash shell offers improved features over the older Bourne and Korn shells and is fully backward-compatible with the Bourne shell. Additionally, the Bash shell, when compared to the other shells, has a more powerful programming interface. As a result, you should use the Bash shell (as contained in Red Hat's Linux version 5.2) to complete your assignment.

Now that you have selected the shell, you need to learn about what the shell scripts include, such as the variables, operators, and special characters.

Variables

Variables are symbolic names that represent values stored in memory. The three types of variables discussed in this section are configuration variables, environment variables, and shell variables. Use **configuration variables** to store information about the setup of the operating system, and do not change them. You can set up **environment variables** with initial values that you can change as needed. These variables, which UNIX reads when you log in, determine many characteristics of your session. For example, you have already learned about the PS1 environment variable, which determines the way your prompt appears. Additionally, UNIX uses environment variables to determine such things as where it should look for programs, which shell to use, and the path of your home directory. **Shell variables** are those you create at the command line or in a shell script. They are very useful in shell scripts for temporarily storing information.

In this chapter you will learn several commands to create, access, and manipulate variables. For example, to see a list of your environment variables, use the **printenv** command.

To see a list of your environment variables:

1. The list of environment variables probably spans more than one screen, so use the more command with printenv. Type **printenv | more** and press **Enter**.

   You see a list of environment variables similar to the one shown in Figure 6-2.

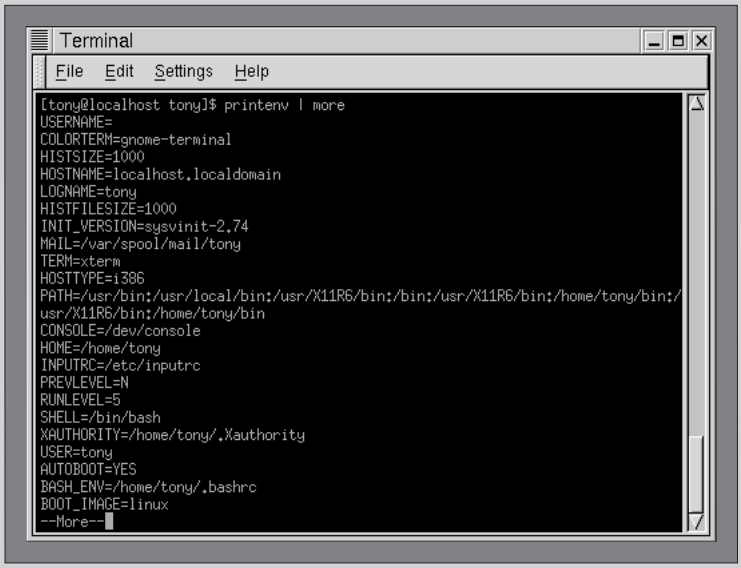

Figure 6-2: Output of printenv | more command

2. Press the **spacebar** until the output is complete.

Note: If you specify the name of a variable as an argument to the printenv command, it only displays the contents of that variable.

Environment and configuration variables bear standard names such as PS1, HOME, PATH, SHELL, USERNAME, and PWD. (Configuration and environment variables are capitalized to distinguish them from user variables). A script file sets the initial values of environment variables in your home directory. You can use these variables to set up and personalize your login sessions. For example, you can set your PATH variable to search for the location of commands that other users have created. Table 6-2 lists standard Bash shell variables.

Name	Variable Contents	Determined by
HOME	Path name for user's home directory	System
LOGNAME	Login name	System
TZ	Time zone used by system	System
SHELL	Path name of program for type of shell you are using	Redefinable
PATH	List of path names for directories searched for executable commands	Redefinable
PS1	Primary shell prompt	Redefinable
PS2	Secondary shell prompt	Redefinable
PS3 and PS4	Holds prompts used by the set and select commands	Redefinable
IFS	Field delimiter symbol	Redefinable
MAIL	Name of mail file checked by mail utility for received messages	Redefinable
MAILCHECK	Interval for checking for received mail (example: 60)	Redefinable
MAILPATH	List of mail files to be checked by mail for received messages	User-defined
TERM	Terminal name	User-defined
MAILWARNING	If set and if a file in which Bash is checking for mail has been accessed since the last time it was checked, the message, "The mail in Mailfile has been read," appears	User-defined
CDPATH	Path names for directories searched by cd command for subdirectories	User-defined
EXINIT	Initialization commands for Ex/vi editor	User-defined
PWD	Current working directory	User-defined

Table 6-2: Standard Bash Shell Environment and Configuration variables

Name	Variable Contents	Determined by
OLDPWD	Previous working directory as set by the cd command	User-defined
REPLY	Set to the line of input read by the read built-in command when no arguments are supplied	User-defined
UID	Expands to the user ID of the current user, initialized at shell startup	User-defined
EUID	Expands to the effective user ID of the current user, initialized at shell startup	User-defined
BASH	Full path name of Bash	User-defined
BASHVERSION	Version number of Bash	User-defined
SHLVL	Incremented by one each time an instance of Bash is started	User-defined
RANDOM	Generates a random integer when referenced. The sequence of random numbers may be initialized by assigning a value to RANDOM.	User-defined
HISTCMD	Index in the history list of the current command	User-defined
HISTSIZE	Number of commands the History command remembers. Default is 500.	User-defined
OPTARG	Value of the last option argument processed by the getopts built-in command	User-defined
OPTFIND	Index of the next argument to be processed by the getopts built-in command	User-defined
HOSTTYPE	Automatically set to a string that uniquely describes the type of machine on which Bash is executing	User-defined
HOSTFILE	Complete host name of the system. In the same format at /etc/hosts.	User-defined

Table 6-2: Standard Bash Shell Environment and Configuration variables (*continued*)

Name	Variable Contents	Determined by
OSTYPE	Description of the operating system on which Bash is executing	User-defined
ENV	Filename containing commands to initialize the shell, as in .bashrc or .tcshrc	User-defined
HISTFILESIZE	Maximum number of lines contained in the history file. Causes the history file to be truncated when this size limit is reached.	User-defined
OPTERR	If set to the value 1, Bash displays error messages generated by getopts command	User-defined
PROMPTCOMMAND	Holds the command to be executed prior to displaying a primary prompt	User-defined
TMOUT	Number of seconds to wait for input after the primary prompt. Bash waits for that number of seconds and terminates if input does not arrive.	User-defined
FCEDIT	Default editor for the fc command	User-defined
FIGNORE	Colon-separated list of suffixes to ignore when performing filename completion	User-defined
INPUTRC	Filename for the Readline start-up file overriding the default of ~/.inputrc	User-defined

Table 6-2: Standard Bash Shell Environment and Configuration variables (*continued*)

Operators are used to define and evaluate variables. Let's take a look at the various operators that UNIX uses.

Shell Operators

The Bash shell operators are divided into three groups: defining and evaluating operators, arithmetic operators, and redirecting and piping operators.

Defining and Evaluating Operators

Use the **equal sign (=) operator** to set a value in a variable, as follows:

```
NAME=Becky
```

This example sets the NAME variable to the value Becky. The variable names or values that appear to the left and right of an operator are its **operands**. The name of the variable you are setting must appear to the left of the = operator. The value of the variable you are setting must appear to the right.

Note: Notice there are no spaces between the = operator and its operands.

To create a variable and assign it a value:

- Type **DOG=Poodle** and press **Enter**.
  You created the variable DOG and set its value to Poodle.

The echo command is used to display the contents of a variable.

To see the contents of a variable:

1. Type **echo DOG** and press **Enter**.
   You see the word "DOG."
2. To see the contents of the DOG variable, you must precede the name of the variable with a $ operator. Type **echo $DOG** and press **Enter**.
   You see the word "Poodle."

Enclose the value in double quotes ("") if it contains spaces. The next command is an example.

To set a variable to a string of characters containing spaces:

1. Type **MEMO="Meeting will be at noon today"** and press **Enter**.
2. Type **echo $MEMO** and press **Enter**.
   You see the contents of the MEMO variable: Meeting will be at noon today.

You may also use single quotes. However, they suppress the evaluation of a variable like $HOME, whereas double quotes do not.

To demonstrate the " and ' characters used with a variable name:

1. Type **echo '$HOME'** and press **Enter**.
   You see $HOME on the screen.
2. Type **echo "$HOME"** and press Enter.
   You see the path of your home directory on the screen.

Finally, the **backquote (`) operator** is used to enclose UNIX commands whose output will become the contents of a variable.

To demonstrate the backquote operator:

1. Type **TODAY=`date`** and press **Enter**. This command creates the variable TODAY, executes the date command, and stores the output of the date command in the variable TODAY. (No output appears on the screen.)
2. Type **echo $TODAY** and press **Enter**. You see the output of the date command that was executed in Step 1.

Exporting Shell Variables to the Environment

Shell scripts cannot automatically access variables created on the command line or by other shell scripts. To make a variable available to a shell script, you must use the export command to make it an environment variable.

To demonstrate the export command:

1. Type **cat > testscript** and press **Enter**.
2. Type **echo $MY_VAR** and press **Enter**.
3. Type **Ctrl+D**. You have created a simple shell script named testscript. Its only function is to display the value of the MY_VAR variable.
4. To make the script executable, type **chmod ugo+x testscript** and press **Enter**.
5. After the $ prompt, type **MY_VAR=2** and press **Enter**.
6. Type **echo $MY_VAR** and press **Enter** to confirm the operation above. You see "2" on the screen.
7. Next look at the list of environment variables. Type **printenv | more** and press **Enter**.
   Look carefully as you scroll through the output of the printenv command. You do not see the MY_VAR variable.
8. Execute the shell script by typing **./testscript** and pressing **Enter**.
   The script displays a blank line. This is because it does not have access to the shell variable MY_VAR.

9. Make the variable available to the script by typing **export MY_VAR** and pressing **Enter**.
10. Execute the script again by typing **./testscript** and pressing **Enter**. This time the value 2 appears.
11. Now look at your list of environment variables by typing **printenv | more** and pressing **Enter**. Again, look carefully as you scroll through the list. This time you see MY_VAR listed.

Notice that in the previous exercise you had to type the ./ characters before the name of the script file. This is because the shell looks for programs in the directories whose names are specified by the PATH variable.

To see the contents of the PATH variable:

- Type **echo $PATH** and press **Enter**.

  You see a list of directories. Notice that the path names are separated by colons (:).

Because new shell programs are most often kept in the current directory while they are being tested, you should add the current working directory to the PATH variable. Here is an example:

```
PATH=$PATH:.
```

Remember, the shell interprets $PATH as the contents of the PATH variable. The sample command sets the PATH variable to its current contents. The colon and dot (.) add the current directory to the search path so that the shell program can locate the new program.

To add the current working directory to the PATH variable:

1. Type **PATH=$PATH:.** and press **Enter**.
2. Type **echo $PATH** and press **Enter**. The dot (.) is now appended to the list.
3. You can now run scripts in your current working directory without typing the ./ characters before their names. Test this by typing **testscript** and pressing **Enter**. You see testscript execute.

Arithmetic Operators

Arithmetic operators consist of the familiar plus (+) for addition, minus (-) for subtraction, asterisk (*) for multiplication, and slash (/) for division. Table 6-3 explains other operators.

Operator	Description	Example
-, +	Unary minus and plus	+R (denotes positive R) -R (denotes negative R)
!, ~	Logical and bitwise negation	!Y (returns 0 if Y is non-zero, returns 1 if Y is zero.) ~X (reverses the bits in X)
*, /, %	Multiplication, division, and remainder	A * B (returns A times B) A / B (returns A divided by B) A % B (returns the remainder of A divided by B)
+, -	Addition, subtraction	X + Y (returns X plus Y) X – Y (returns X minus Y)
>, <	Greater than and less than	M > N (Is M greater than N?) M < N (Is M less than N?)
=, !=	Equality and inequality	Q = R (Is Q equal to R?) Q != R (Is Q not equal to R?)

Table 6-3: Examples of the shell's arithmetic operators

When using arithmetic operators, the usual mathematical precedence rules apply: multiplication and division are performed before addition and subtraction. For example, the value of the expression 6 + 4 * 2 is 14, not 20. Precedence can be overridden, however, by using parentheses. For example, the value of the expression (6 + 4) * 2 is 20, not 14. Other mathematical rules also apply; for example, division by zero is treated as an error.

To store arithmetic values in a variable, use the let statement. For example, the following command stores 14 in the variable X:

```
let X=6+4*2
```

You can use shell variables as operands to arithmetic operators. Assuming the variable X has the value 14, the following command stores 18 in the variable Y:

```
let Y=X+4
```

To practice using the arithmetic operators:

1. Type **let X=10+2*7** and press **Enter**.
2. Type **echo $X** and press **Enter**. You see 24 on the screen.
3. Type **let Y=X+2*4** and press **Enter**.
4. Type **echo $Y** and press **Enter**. You see 32 on the screen.

Note: Constants beginning with 0 are interpreted as octal numbers. A preceding x denotes a hexadecimal number.

Preventing Redirection from Overwriting Files

Recall that the > redirection operator overwrites an existing file. If you write a shell script that uses the > operator to create a file, you may want to prevent it from overwriting important information. You can use the set command with the **–o noclobber option** to prevent a file from being overwritten, as in the following example:

```
$ set –o noclobber     <Enter>
```

However, you can choose to overwrite a file anyway by placing an exclamation point after the redirection operator:

```
$ set –o noclobber     <Enter>
$ cat new_file > old_file
$ cat: file exists
$ cat: new_file >! old_file <Enter>
```

More About Wildcard Characters

Shell scripts frequently use the asterisk (*) and other wildcard characters (such as ? and []), which help to locate information containing only a portion of a matching pattern. For example, to retrieve all program files whose names contain a ".c" extension, use the following command:

```
ls *.c
```

Wildcard characters are also known as **glob** characters. If an unquoted argument contains one or more glob characters, the shell processes the argument for filename generation. Glob characters are part of **glob patterns,** which are intended to match filenames and words. Special constructions that may appear in glob patterns follow:

- The question mark (?) matches exactly one character, except for the backslash and period.
- The asterisk (*) matches zero or more characters in a filename.
- [chars] defines a class of characters. The glob pattern matches any single character in the class. A class may contain a range of characters, as in [a-z].

For example, assume the working directory contains files chap1, chap2, and chap3. The following command displays the contents of all three files:

```
more chap[1–3]     <Enter>
```

Now that you understand some of the tools used to construct shell scripts, you'll see examples of how shell scripts are written to support the four basic logic structures needed for program development.

Shell Logic Structures

The four basic logic structures needed for program development are sequential logic, decision logic, looping logic, and case logic.

Sequential Logic

Sequential logic states that commands will be executed in the order in which they appear in the program. The only break in this sequence comes when a branch instruction changes the flow of execution. The following steps present a very simple example of a shell script using sequential logic. You create a simple shell script, seqtotal.

To demonstrate sequential logic:

1. Type **vi seqtotal** and press **Enter**.
2. Type **i** to switch to vi's insert mode.
3. Type the following lines:

```
let a=1
let b=2
let c=3
let total=a+b+c
echo $total
```

4. Press **Esc** to switch to vi's command mode.
5. Type **:wq** and press **Enter** to save the file and exit vi.
6. Next, test the new shell script, Seqtotal. (To save a few keystrokes, you use the sh command instead of the chmod command.) Type **sh seqtotal** and press **Enter**.

   You see the output of the program, which is 6.

Many programs are simple, straightforward command sequences. An example is the programmer activity report script you wrote in Chapter 5. The shell executes the script's commands in the order they appear in the file. You use sequential logic to write this type of application.

Decision Logic

Decision logic enables your program to execute a statement or series of statements only if a certain condition exists. The **if statement** is the primary decision-making control structure in this type of logic. In the steps that follow, notice that the semicolon (;) separates commands on the same line.

Note: Throughout the sample scripts, variables are always enclosed in double quotes, as in "$choice," "$looptest," "$yesno," "$guess," and "$myfavorite," due to how the shell interprets variables. All shell variables, unless declared otherwise, are strings, which are arrays of characters. If you do not enter data in the string variables, the variables will be treated as blank strings, which result in an invalid test. The enclosing double quotes therefore maintain the validity of strings, with or without data, and the test is carried out without producing an error condition.

To demonstrate decision logic:

1. Type **vi os_choice** and press **Enter**.
2. Type **i** to switch to vi's insert mode.
3. Type the following lines:

```
echo -n "What is your favorite operating system? "
read OS_NAME
if [ "$OS_NAME" = "UNIX" ]
then
   echo "You will like Linux."
else
   echo "You should give Linux a try!"
fi
```

4. Press **Esc** to switch to vi's command mode.
5. Type **:wq** and press **Enter** to save the file and exit vi.

Before you run the program, let's examine its contents. The first statement uses the echo command to display a message on the screen. The –n option suppresses the line feed that normally appears after the message. The second statement uses the read command, which waits for the user to type a line of keyboard input. The input is stored in the variable specified as the read command's argument. The line in the program reads the user's input into the OS_NAME variable.

The next line begins an if statement. The word "if" is followed by an expression inside a set of brackets ([]). (The spaces that separate the [and] characters from the enclosed expression are necessary.) The expression, which is tested to determine if it is true or false, compares the contents of the OS_NAME variable with the string UNIX. (When you use the = operator in an if statement's test expression, it tests its two operands for equality. If the operands are equal, the expression is true. Otherwise, it is false.) If the contents of the OS_NAME variable is equal to UNIX, the statement that follows the word "then" is executed. In this program, it is an echo statement.

If the if statement's expression is false (if the contents of the OS_NAME variable does not equal UNIX), the statement that follows the word "else" is executed. In this program, it is a different echo statement.

Notice the last statement, which is the word fi. fi ("if" spelled backward) always marks the end of an if or an if...else statement.

Now, test your code by executing it at least twice:

1. Run the program.
2. When asked to enter the name of your favorite operating system, answer **UNIX**.
3. Run the program again and respond with **MS-DOS** or some other operating system name.

Remember to use the chmod command first to make the script executable. Then type the script's name after the command prompt to execute it. Another way to run the script is to use the sh command, as you did with the seqtotal script.

Nested Control Structures You can nest a control structure, such as the if statement, inside another control structure. For example, a program may have an if statement inside another if statement. The first if statement controls when the second if statement is executed.

To practice writing a nested if statement:

1. Open the os_choice file in vi or Emacs.
2. Edit the file so it contains the following lines. (Code has been added to the else part of the original if statement.)

```
echo -n "What is your favorite operating system? "
read OS_NAME
if [ "$OS_NAME" = "UNIX" ]
then
  echo "You will like Linux."
else
  if [ "$OS_NAME" = "WINDOWS" ]
  then
       echo "A great OS for applications."
  else
       echo "You should give Linux a try!"
  fi
fi
```

3. Execute the program and respond with **WINDOWS** when asked for your favorite operating system.

As you can see, the second if statement is located in the first if statement's else section. It is only executed when the first if statement's expression is false.

Decision logic structures, such as the if statement, are used in applications where different courses of action are required, depending on the result of a command or comparison.

Looping Logic

In **looping logic**, a control structure (or loop) repeats until some condition exists or some action occurs. You will learn two looping mechanisms in this section: the for loop and the while loop.

You use the **for** command for looping through a range of values. It causes a variable to take on each value in a specified set, one at a time, and perform some action while the variable contains each individual value. The loop stops after the variable has taken on the last value in the set and has performed the specified action with that value.

To demonstrate looping logic:

1. Create the file our_users with vi or Emacs.
2. Type the following lines into the file:

```
for USERS in john ellen tom becky eli jill
do
  echo $USERS
done
```

3. Save the file and exit the editor.
4. Give the file execute permission, and run it. Your screen appears similar to Figure 6-3.

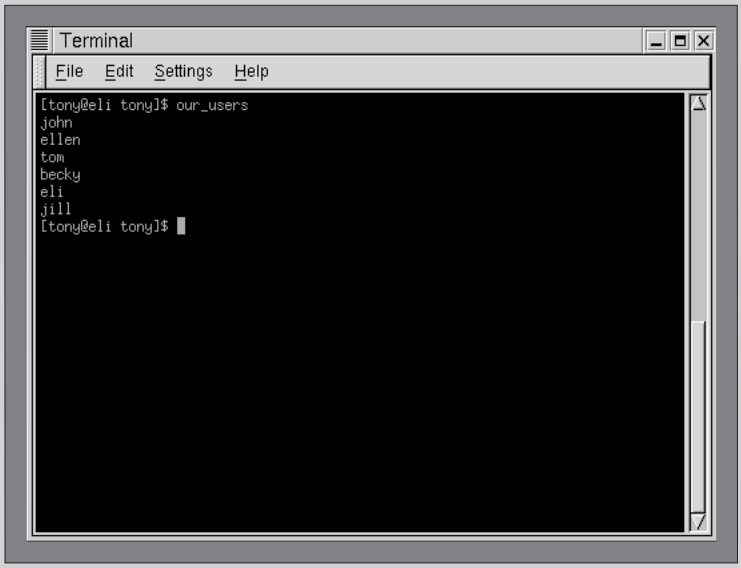

Figure 6-3: Output of Our_users script

The for statement you typed in Step 2 specifies that the variable USERS take on the values john, ellen, tom, becky, eli, and jill. Because there are six values in the set,

the loop repeats six times. Each time it repeats, USERS contains a different value from the set, and the statement between the do and done statements is executed. The first value in the set is john, so the echo statement executes the first time with john stored in USERS. That is why john appears on the screen. The second time the loop executes, USERS contains ellen, so the echo statement displays ellen. This procedure repeats until no other names are left in the set.

Executing Control Structures at the Command Line Most shell script control structures, such as the if and for statements, must be written across several lines. This does not prevent you from executing them directly on the command line, however.

To demonstrate entering program control structures at the command line:

- Enter the following. Note you will type directly from the command line and not into a script file.

```
$ for myhobbies in tennis swimming movies travel <Enter>
> do                                              <Enter>
> echo $myhobbies                                 <Enter>
> done                                            <Enter>
```

After you finish typing you see results similar to Figure 6-4.

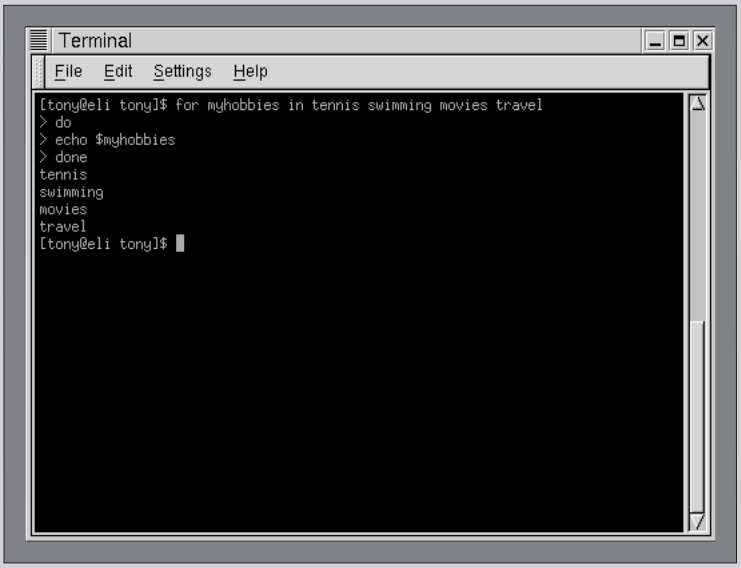

Figure 6-4: Results of for loop typed at command line

The shell knows more code will come after you type the first line. It displays the > prompt indicating it is ready for the control structure's continuation. The shell reads further input lines until you type the word "done," which marks the end of a for loop.

Using Wildcard Characters in a Loop Now you can see how the for statement works with wildcard characters when you want to print a few chapters of a book.

To create test data and use wildcards in a for loop:

1. Type **cat > chap1** and press **Enter**.
2. Type **This is chapter 1** and press **Enter**.
3. Type **Ctrl+D**. The file chap1 is created.
4. Type **cat > chap2** and press **Enter**.
5. Type **This is chapter 2** and press **Enter**.
6. Type **Ctrl+D**. The file chap2 is created.
7. Type **cat > chap3** and press **Enter**.
8. Type **This is chapter 3**, and press **Enter**.
9. Type **Ctrl+D**. The file chap3 is created.
10. Use the vi or Emacs editor to create the shell program, chapters. The program should have these lines:

```
for file in chap[123]; do
  more $file
  done
```

11. Save the file and exit the editor.
12. Give the file execute permission, and test it. You see output similar to Figure 6-5.

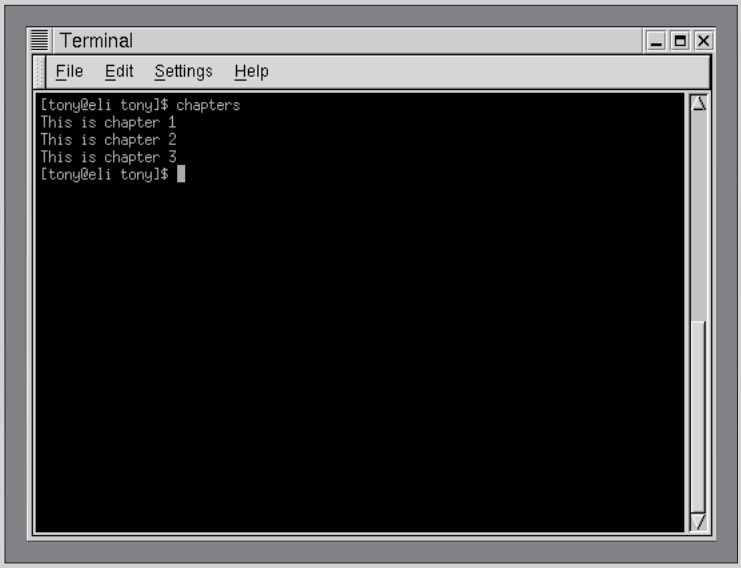

Figure 6-5: Results of chapters script

The While Loop A different pattern for looping is created using the while statement. The **while** statement best illustrates how to set up a loop to test repeatedly for a matching condition.

To use the while statement:

1. Use the vi or Emacs editor to create a shell script, colors.
2. The script should contain these lines:

```
echo -n "Try to guess my favorite color "
read guess
while [ "$guess" != "red" ]; do
  echo "No, not that one. Try again. "; read guess
done
```

3. Save the file and exit the editor.
4. Give the file execute permission, and test it. You see output similar to Figure 6-6.

Figure 6-6: Results of colors script

The while loop tests an expression in a manner similar to the if statement. As long as the statement inside the brackets is true, the statements inside the do and done statements repeat.

Notice the use of the != symbol, which is the not-equal operator. It tests two operands and returns true if they are not equal. Otherwise, it returns false. In the steps you just completed, the echo and read statements inside the loop repeat until the user enters red.

Another example of the while loop is in a data entry form.

To enter a while loop that will serve as a data entry form:

1. Use vi or Emacs to create a script file, nameaddr.
2. Type these lines into the file:

```
looptest=y
 while [ "$looptest" = y ]
  do
     echo –n "Enter Name  : "; read name
     echo –n "Enter Street : "; read street
     echo –n "Enter City  : "; read city
     echo –n "Enter State  : "; read state
     echo –n "Enter Zip Code: "; read zip
     echo –n "Continue? (y)es or (n)o "; read looptest
  done
```

3. Save the file and exit the editor.
4. Give the file execute permission, and test it. You see output similar to Figure 6-7.

Figure 6-7: Output of nameaddr script

As you test the program, enter several names and addresses. When you finish, answer n (for no) when the program asks you, "Continue? (y)es or (n)o."

Use looping logic in applications where code must be repeated a determined or undetermined number of times.

Case Logic

The **case logic** structure simplifies the selection of a match when you have a list of choices. It allows your program to perform one of many actions, depending upon the value of a variable. Note the use of two semicolons (;;) that terminate the action(s) taken after the case matches to the test.

To demonstrate case logic:

1. Use the vi or Emacs editor to create the manycolors shell script.
2. Type these lines into the file:

```
echo –n "Enter your favorite color: "; read color
 case "$color" in
   "blue")  echo "As in My Blue Heaven.";;
   "yellow") echo "As in the Yellow Sunset.";;
   "red")  echo "As in Red Rover, Red Rover.";;
   "orange") echo "Autumn has shades of Orange.";;
   *   ) echo "Sorry, I do not know that color.";;
esac
```

3. Save the file and exit the editor.
4. Give the file execute permission, and test it. You see output similar to that shown in Figure 6-8.

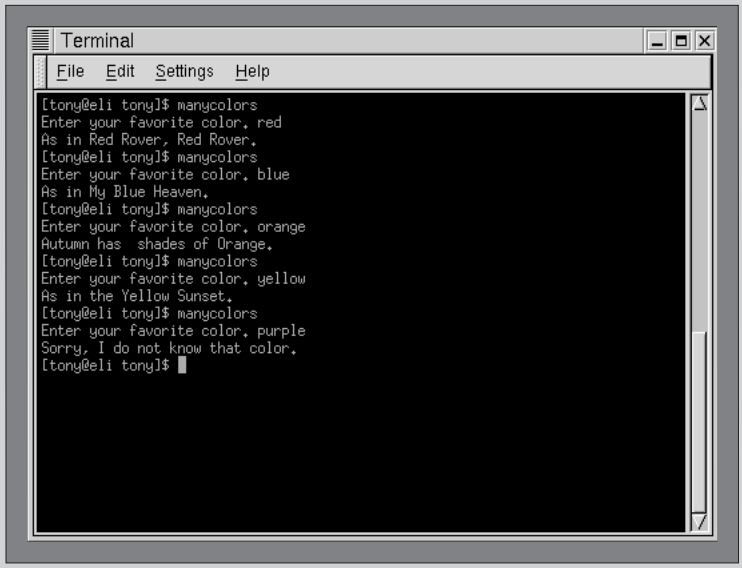

Figure 6-8: Output of manycolors script

In the Manycolors script, the case structure examines the contents of the color variable and searches for a match among the values listed. When a match is found, the statement that immediately follows the case value is executed. For example, if the color variable contains red, the echo statement that appears after "red") is executed. If the contents of the colors variable does not match any of the values listed, the statement that appears after *) is executed.

As you can see, case logic is designed to pick one course of action from a list of many, depending upon the contents of a variable. This control structure is ideal for menus, where the user chooses one of several values.

Note: The case structure is terminated by the word "esac," which is "case" spelled backwards.

In this section, you learned about shell variables, operators, and wildcard characters. You also learned about the four logic structures. In the next section, you'll use shell programming to create your application.

SUMMARY

- A high-level language (such as C, C++, or COBOL) is a language that uses English-like expressions. A high-level language must be converted into a low-level (machine) language before the computer can execute it.
- The shell interprets UNIX shell scripts. They do not need to be converted to machine language because UNIX converts and executes them in one step.
- UNIX shell scripts, created with the vi or Emacs editor, contain instructions that do not need to be written from scratch, but rather can be selectively chosen from the operating system's inventory of executable commands.
- Linux shells are derived from the UNIX Bourne, Korn, and C shells. The three Linux shells are Bash, tcsh, and zsh; Bash is the most used Linux shell.
- UNIX keeps three types of variables: configuration, environment, and shell. Configuration variables contain set-up information for the operating system. Environment variables keep information about your login session. Shell variables are created in a shell script or at the command line. The export command is used to make a shell variable an environment variable.
- The shell supports numerous operators, including many for performing arithmetic operations.
- You can use wildcard characters in shell scripts, including the bracket ([]) characters. Brackets surround a set of values that can match an individual character in a name or string.
- The logic structures supported by the shell are sequential, decision, looping, and case.

REVIEW QUESTIONS

1. Shell programs _____.
   a. can be developed faster than compiler programs
   b. run slower than their compiler counterparts
   c. serve as excellent prototypes
   d. All of the above
2. What is the difference between a configuration variable and an environment variable?
   a. The user sets configuration variables.
   b. The system sets environment variables.
   c. The system sets configuration variables.
   d. The user should never change environment variables.
3. Which of the following commands protect a file from being overwritten?
   a. set –o noclobber
   b. chmod 777 *filename*
   c. set –o clobber
   d. set !*filename*
4. The four basic logic structures are _____.
   a. sequence, decision, loop, and case
   b. sequence, iteration, loop, and case
   c. decision, iteration, case, and if-then
   d. for, next, while, and if
5. Which of these commands stores the output of the date command in the variable MYDATE?
   a. MYDATE=date
   b. date > MYDATE
   c. MYDATE=`date`
   d. date | MYDATE
6. The program development cycle can be described as _____.
   a. program design, program specifications, writing code, testing and debugging
   b. program specifications, program design, writing code, testing and debugging
   c. writing code, testing and debugging, program design, program specifications
   d. program design, writing code, testing and debugging, program specifications
7. The _____ command displays a list of environment variables.
   a. printenv
   b. envars
   c. ls –env
   d. cat envars
8. The _____ command makes a variable an environment variable.
   a. send
   b. port
   c. export
   d. import

9. Sequential logic can be described as _____.
   a. a logic structure for executing a statement or group of statements only if a certain condition exists
   b. a logic structure for executing a series of statements in the order they are written
   c. a logic structure for repeating a statement or series of statements
   d. a logic structure for performing one of many actions, depending upon the value of a variable

10. Decision logic can be described as _____.
   a. a logic structure for executing a statement or group of statements only if a certain condition exists
   b. a logic structure for executing a series of statements in the order they are written
   c. a logic structure for repeating a statement or series of statements
   d. a logic structure for performing one of many actions, depending upon the value of a variable

11. Looping logic can be described as _____.
   a. a logic structure for executing a statement or group of statements only if a certain condition exists
   b. a logic structure for executing a series of statements in the order they are written
   c. a logic structure for repeating a statement or series of statements
   d. a logic structure for performing one of many actions, depending upon the value of a variable

12. Case logic can be described as _____.
   a. a logic structure for executing a statement or group of statements only if a certain condition exists
   b. a logic structure for executing a series of statements in the order they are written
   c. a logic structure for repeating a statement or series of statements
   d. a logic structure for performing one of many actions, depending upon the value of a variable

13. The _____ command stores the user's input in a variable.
   a. input
   b. get
   c. read
   d. import

14. Shell programs are _____.
   a. compiled
   b. interpreted
   c. both compiled and interpreted
   d. neither compiled nor interpreted

EXERCISES

1. Create the file alphalist with the following entries:

   Alpha_romeo

   Chevrolet

   Buick

   Plymouth

   Dodge

   BMW

2. Sort the alphalist file and pipe it to a cat program to display a numbered list of cars.
3. Create a new file, myfile, in your home directory.
   a. Use the set –o noclobber. Create a second file, mynewfile.
   b. Enter this command: cat mynewfile > myfile <Enter>.
   c. Explain what happened.
   d. Now modify the redirection to make it work.
4. Fix this line of code to make it work:

   ```
   echo "Why don't you get a "van down by the river?"
   ```

5. If you do not have a bin directory in your home directory, create one. Look at the contents of the PATH variable. If it does not include the path to your bin directory, add it.
6. Create a variable, NAME. Store your full name in it.
7. Use the printenv command to display a list of environment variables. Is the NAME variable listed? Execute the command that makes NAME an environment variable.

DISCOVERY EXERCISES

1. Use the sample menu as a guide, and create a new menu program that looks like this:

   Travel Menu

   ==========

   (L)ondon scenic tour

   (P)aris in the Spring

   (N)ew York Shopping Fantasy

   Make your selection … (Q) to quit

2. Enter the command to store the output of the pwd command in a variable, PLACE. Display the contents of the PLACE variable on the screen.
3. Write a script named change_prompt. The script should change your prompt to display the date and the name of your working directory. Give the file execute permission, and test it.

4. Execute each of these commands:

```
let X=2+4*3
Y=2+4*3
```

   Display the contents of each variable on the screen. How do you explain the difference?

5. Write a script file that performs the following steps:
   a. Stores 5 in variable A
   b. Stores 10 in variable B
   c. Stores the sum of A plus B in variable C
   d. Stores the difference of B minus A in variable D
   e. Stores the product of A times B in variable E
   f. Stores the quotient of B divided by A in variable F
   g. Displays the contents of all the variables

6. Write a script that asks the user to enter his or her first name. The script should include an if statement that compares the name the user entered with your name. If they are the same, the script should display the message, "That is my name too." Otherwise it should display the message, "That is a nice name."

7. Write a script file with a for loop that displays a list of your favorite music groups.

8. Repeat Discovery Exercise 7, but this time enter the for loop at the command line.

9. Write a script file that asks users to guess your name. If they guess incorrectly, the script should ask them to guess again. Repeat this until users correctly guess your name. (HINT: Use the while loop.)

10. Write a script file that asks the user to choose and enter the name of a country from the following list: America, Italy, France, Germany. The script then uses a case structure to display the name of a car made in the selected country. Use the cars listed in the table.

Country	Car
America	Ford
Italy	Ferrari
France	Peugeot
Germany	Porsche

LESSON B

Objectives

In this lesson you will:

- Create screen-management and cursor programs
- Edit the .bashrc file to customize your personal working environment
- Use the trap command
- Enter and test shell programs to print the phone records, view the contents of the Corp_phones file, and add new phone records to the file

Creating and Completing the Corporate Phones Application

Using Shell Programming to Create a Menu

Before creating your corporate phone program, you first create a menu, which is a good example of a shell program that employs the four basic logic structures: sequence, decision, looping, and case. A significant feature of the menu program is the screen presentation, which you want be as appealing and user-friendly as possible. You can choose one of UNIX's lesser-known commands, tput, to make your screen place the prompt (cursor) at the user's data-entry point on the screen. The **tput command** initializes the terminal to respond to a setting that the user chooses. Some examples of this are:

- tput cup 0 0: moves the cursor to row 0, column 0, the upper-left corner of screen
- tput clear: clears the screen
- tput cols: prints the number of columns for the current terminal
- bold=`tput smso` offbold=`tput rmso`: sets boldfaced type

To practice the tput command:

- Type the following command sequence, and press **Enter**:

```
tput clear ; tput cup 10 15 ; echo "Hello" ; tput cup 20 0
```

You see the results of the command sequence. The screen clears; the cursor is positioned at row 10, column 15, on the screen; the word "Hello" is printed; and the prompt's position is row 20, column 0.

Next, use the tput command to create a cursor utility. You can use this utility in other script files to position the cursor on the screen.

To use the tput command to create a cursor utility:

1. Use the vi or Emacs editor to create a script file, cursor.
2. Enter the following lines in the program:

```
tput cup $1 $2
```

3. Save the file and exit the editor.
4. Make the cursor script executable with the chmod command.
5. Test the cursor script by typing **cursor 10 12** and pressing **Enter**. The cursor moves to the position 10, 12 on the screen.

Notice the script is passing $1 and $2 as arguments to the tput cup command. These are positional parameters: they reference arguments typed on the command line when the cursor script file is executed. For example, when you invoked the script with the command cursor 10 12, the $1 parameter holds the value 10, and the $2 parameter holds the value 12.

You will now use the tput command to create screen management and cursor programs for displaying menus and data entry screens.

To use the tput command:

1. Use the vi or Emacs editor to enter a screen-management program, Scrmanage, containing the following lines:

```
tput cup $1 $2  # place cursor on row and col
tput clear    # clear the screen
bold=`tput smso` # set standout mode - bold
offbold=`tput rmso` # reset screen – turn bold off
echo $bold     # turn bold on
cursor 10 20; echo "Type Last Name:" # bold caption
cursor 12 20; echo "Type First Name:" # bold caption
echo $offbold   # turn bold off
cursor 10 41; read lastname # enter last name
cursor 12 41; read firstname # enter last name
```

2. Save the file and exit the editor.
3. Give the file execute permission, and test it.

You are now ready to enter the script to display the menu. (You will use this program as part of your application later in this section.)

To enter the script:

1. Use the vi or Emacs editor to enter the Phmenu program shown below.

```
#=========================================================
  =
# Script Name:  phmenu
# By:           JQD (Your initial here)
# Date:         November 1999
# Purpose:      A menu for the Corporate Phone List
# Command Line: phmenu
#=========================================================
  =
loop=y
while [ "$loop" = y ]
do
  clear
  cursor 3 12; echo "Corporate Phone Reporting Menu"
  cursor 4 12; echo "=============================="
  cursor 6 9; echo "P - Print Phone List  :"
  cursor 7 9; echo "A - Add New Phones    :"
  cursor 8 9; echo "S - Search for Phones :"
  cursor 10 9; echo "Q - Quit"
  cursor 11 32
  read choice || continue
done
```

2. Save the file and exit the editor.

3. Give the file execute permission, and test the program. (You will have to press **Ctrl+C** to exit the program, as the Quit option has not yet been programmed.)

Now that you have an idea of how to create a menu program, it will be helpful to learn some additional shell features and commands before creating your application. You will first learn how to customize your personal environment so that creating and testing a new program do not interfere with your other programs.

Customizing Your Personal Environment

When your work requirements center on computer programming and shell scripting, consider customizing your environment by modifying the initial settings in the login scripts. For example, many programmers set up a personal bin directory where they can store and test their new programs without interfering with ongoing operations. ("Bin" is the traditional UNIX name for directories that hold executable files.) You can also modify the editors to make them automatically indent the text inside their programs in accordance with programmer usage.

An **alias** is a name that represents another command. You can use aliases to simplify and automate commands you use frequently. For example, the following command sets up an alias for the rm command.

```
alias rm="rm –i"
```

This command above causes the rm –i command to execute any time the user enters the rm command. This is a commonly used alias, as it ensures users are always prompted before the rm command deletes a file. Here are two other common aliases that help safeguard files:

```
alias mv="mv –i"
alias cp="cp –i"
```

To practice creating an alias:

1. To create an alias for the ls command, type **alias ls="ls –l"** and press **Enter**. Now, when you use the ls command, the ls –l command executes automatically.
2. Test the alias by typing **ls** and pressing **Enter**. You see a long directory listing.

The **.bashrc file** that resides in your home directory can be used to establish customizations that take effect for each login session. The .bashrc script is executed each time you generate a shell, such as when you run a shell script. Any time a subshell is created, the .bashrc is re-executed. The following .bashrc file is commented to explain how you can make your own changes.

```
.bashrc
# Source global definitions
if [ -f /etc/bashrc ]; then
     . /etc/bashrc # if any global definitions are
                   # run them first
alias rm ‘rm –i'  # make sure user is prompted before
                   # removing files
alias mv 'mv –i'  # make sure user is prompted before
                   # overlaying files
set –o ignoreeof    # Do not allow Ctrl-D to logout

set –o noclobber    # Force user to enter >! To write
           # over existing files
PS1="\w \$"     # Set prompt to show working directory
```

In addition to knowing how to customize your work environment, you should also be familiar with the trap command.

The trap Command

The trap command is useful when you want your shell program to automatically remove any temporary files that are created when the shell scripts runs. The **trap** command specifies that a command, listed as the argument to trap, is read and executed when the shell receives a specified system signal. Here is an example of the command:

```
trap "rm ~/tmp/* 2> /dev/null; exit" 0
```

This command has two arguments: a command to be executed, and a signal number from the operating system. The command, rm ~/tmp/* 2> /dev/null; exit, deletes everything in the user's tmp directory, redirects the error output of the rm command to the null device (so it does not appear on the screen), and issues an exit command to terminate the shell. The signal specified is 0, which is the operating system signal generated when a program exits. So, if this sample command is part of a script file, it causes the specified rm command to execute when signal 0 is sent by the operating system.

The programmer often sets up ~/tmp (a subdirectory of the user's home directory) to store temporary files. When the script file exits, any files placed in ~/tmp can be removed. This is called "good housekeeping" on the part of the programmer.

Creating the Corp_phones File

The steps that follow require you to create the corp_phones file and place it in your /home/user/source directory.

To create the corp_phones file:

1. Use the vi or Emacs editor to create the corp_phones file.
2. Enter the following records in the file:

```
219-555-4567:Harrison:Joel:M:4540:Accountant:09-12-1985
219-555-4587:Mitchell:Barbara:C:4541:Admin Asst:12-14-1995
219-555-4589:Olson:Timothy:H:4544:Supervisor:06-30-1983
219-555-4591:Moore:Sarah:H:4500:Dept Manager:08-01-1978
219-555-4567:Polk:John:S:4520:Accountant:09-22-1998
219-555-4501:Robinson:Albert:J:4501:Secretary:08-12-1997
```

3. Save the file and exit the editor.
4. Use the following grep command to search the file for specific phone numbers. The output should look like:

```
$grep 219-555-4591 corp_phones  <Enter>
219-555-4591:Moore:Sarah:H:4500:Dept Manager:08-01-1978
```

5. Use a similar grep command to search the file for all Accountants. Your screen should look similar to Figure 6-9.

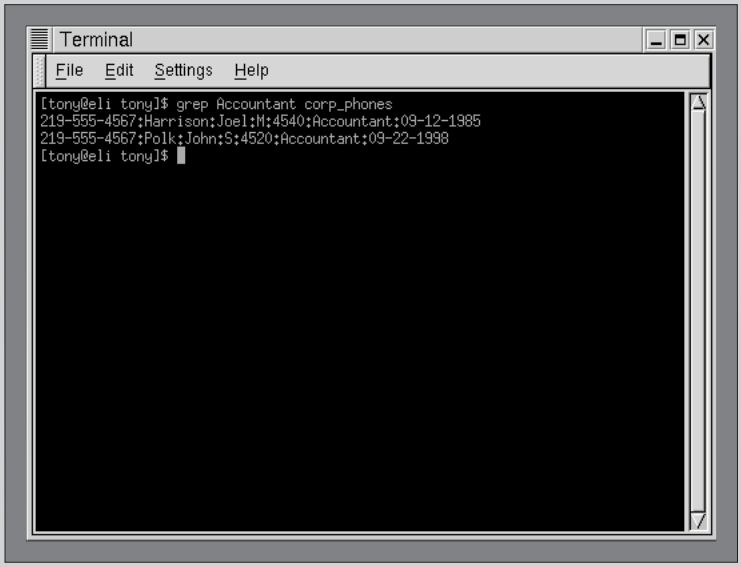

Figure 6-9: Results of grep command

Notice that you can extract information from the corp_phones file using grep, as well as other commands you learned in previous chapters. You will use such commands in building your script application.

You will now complete the menu program that you started at the beginning of this lesson.

To complete the phone menu program:

1 Use the vi or Emacs editor, and retrieve phmenu. Add the boldfaced lines to the script.

```
#=========================================================
# Script Name:   phmenu
# By:            JQD
# Date:          March 1999
# Purpose:       A menu for the Corporate Phone List
# Command Line:  phmenu
#=========================================================
phonefile=~/source/corp_phones
loop=y
while [ "$loop" = y ]
do
  clear
  cursor 3 12; echo "Corporate Phone Reporting Menu"
  cursor 4 12; echo "=============================="
  cursor 6 9; echo "P - Print Phone List  :"
```

```
  cursor 7 9; echo "A - Add Phone to List :"
  cursor 8 9; echo "S - Search for Phone  :"
  cursor 10 9; echo "Q - Quit"
  cursor 11 32;
  read choice || continue
     case $choice in
       [Aa]) phoneadd ;;
       [Pp]) phlist1 ;;
       [Ss]) phonefind ;;
       [Vv]) less $phonefile;;
       [Qq]) exit ;;
       *)  cursor 14 4; echo "Invalid Code"; read prompt ;;
     esac
done
```

2 Save the file and exit the editor.

3 Give the file execute permission, and test it.

When you demonstrate your program to Dominion Consulting managers, they point out that some errors may occur when entering additional information. To locate errors quickly, you need to view unformatted records.

Viewing Unformatted Records

It is often useful to have a program that displays unformatted file data, which means that the records appear exactly as they are stored on the disk. In the case of the phone records, the display shows the colon (:) characters that separate the variable-length fields. If problems develop, it is convenient to have a look at the raw data. To do this, you use the **less command** to display the records on the screen.

To use the less command to view unformatted records:

1 Open Phmenu in the editor of your choice, and add the boldfaced lines shown.

```
#=========================================================
# Script Name:   phmenu
# By:            JQD
# Date:          March 1999
# Purpose:       A menu for the Corporate Phone List
# Command Line:  phmenu
#=========================================================
phonefile=~/source/corp_phones
loop=y
while [ "$loop" = y ]
```

```
do
  clear
  cursor 3 12; echo "Corporate Phone Reporting Menu"
  cursor 4 12; echo "============================="
  cursor 6 9; echo "P - Print Phone List  :"
  cursor 7 9; echo "A - Add Phone to List :"
  cursor 8 9; echo "S - Search for Phone  :"
  cursor 9 9; echo "V - View Phone List   :"
  cursor 10 9; echo "Q - Quit"
  cursor 11 32;
  read choice || continue
     case $choice in
       [Aa]) phoneadd ;;
       [Pp]) phlist1 ;;
       [Ss]) phonefind ;;
       [Qq]) exit ;;
       *)  cursor 14 4; echo "Invalid Code"; read prompt ;;
     esac
done
```

2 Save the file and exit the editor.

3 Test the script.

You are now ready to run a prototype of your program. To do this, you use the Awk program.

Using Awk

The Awk program offers a good example of how the UNIX shell programmer can quickly create an application model. Using Awk accelerates development because a single awk command can select fields from many records and display them in a specified format on the screen. To run a prototype of your corporate phone program, combine the cursor program you wrote earlier with a line of Awk code.

To use the Awk program:

1 Use the editor of your choice to create the **Phlist1** program. Use the following code as a guide.

```
#
#-------------------------------------------------------
# Script Name:   phlist1
# By:            JQD
# Date:          March 1999
```

```
# Purpose:      Use awk to format colon-separated fields
#           in a flat file and display to the screen
# Command line: phlist1
# ========================================================
clear
cursor 2 20; echo "Corporate Phone List"
cursor 3 20; echo "====================="
cursor 5 0;
awk -F: ' { printf "%-12s %-12s %s\t%s-%s-%s %s %10.10s
%s\n", $4, $5, $6, $1, $2, $3, $7, $8, $9 } ' Corp_phones
```

2. Save the file and exit the editor.
3. Give the file execute permission and run it. Your screen should look like Figure 6-10.

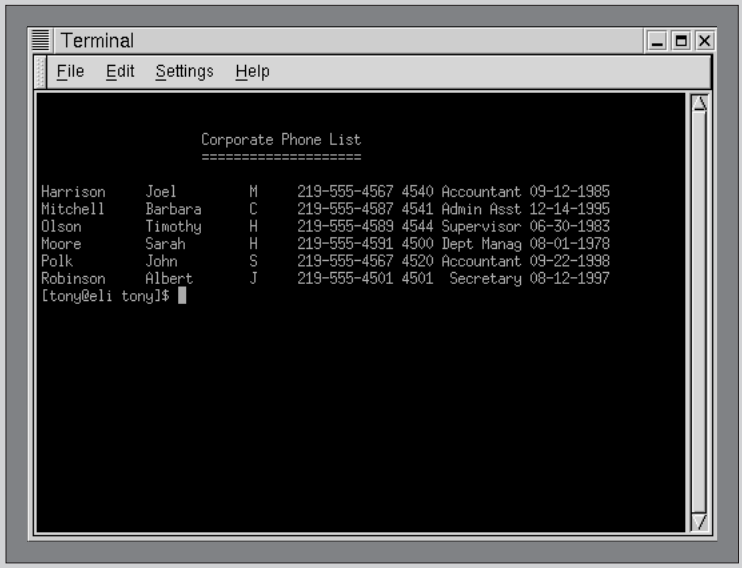

Figure 6-10: Output of Phlist1 script

You realize that names will be added to the corp_phones file, so you need to modify your shell script.

Creating the Phoneadd Shell Program

The Phmenu program offers the option to add a phone record to the corp_phones file; that is, the user interacts with the program to enter a new employee's phone record. To do this, you can use a cursor program, similar to the one you wrote earlier, to design your data entry screen, display it, and place the cursor in the first data-entry field. You can subsequently position it for all the other entries.

To create the phoneadd program to allow additions:

1 Use the editor of your choice to create the script phoneadd. Enter this code:

```
# ========================================================
# Script Name: phoneadd
# By:          JQD
# Date:        March 1999
# Purpose:     A shell script that sets up a loop to add
#              new employees to the Corp_phones file.
#                The code also prevents duplicate phone
#              numbers from being assigned.
# Command Syntax:  phoneadd
#
# ========================================================
trap "rm ~/tmp/* 2> /dev/null; exit" 0 1 2 3
phonefile=~/corp_phones
looptest=y
while [ $looptest = y ]
do
   clear
  cursor 1 4; echo "Corporate Phone List Additions"
  cursor 2 4; echo "============================="
  cursor 4 4; echo "Phone Number: "
  cursor 5 4; echo "Last Name   : "
  cursor 6 4; echo "First Name  : "
  cursor 7 4; echo "Middle Init : "
  cursor 8 4; echo "Dept #      : "
  cursor 9 4; echo "Job Title   : "
  cursor 10 4; echo "Date Hired  : "
  cursor 12 4; echo "Add Another? (Y)es or (Q)uit "
   cursor 4 18; read phonenum
  if [ "$phonenum" = 'q' ]
       then { clear; exit }
  fi
  cursor 5 18; read lname
  cursor 6 18; read fname
      cursor 7 18; read midinit
      cursor 8 18; read deptno
       cursor 9 18; read jobtitle
      cursor 10 18; read datehired
      # Check to see if last name is not a blank before you
      # write to disk
      if [ "$lname" > "        " ]
      then
echo "$phonenum:$lname:$fname:$midinit:$deptno:$jobtitle: $date hired" >> $phonefile
      fi
```

```
  cursor 14 33; read looptest
  if [ "$looptest" = 'q' ]
    then { clear; exit }
  fi
done
```

2. Save the file and exit the editor. Give the file execute permission and run it. Test the program by adding the following employees:

```
219-555-7175 Mullins  Allen L 7527 Sales Rep 09-12-2000
219-555-7176 Albertson Jeannette K 5547 DC Clerk
09-12-2000
```

3. Now run the phmenu script and select **A** from the menu to execute the phone and script. Add another record to the file and then quit the program.

Your screen should look similar to Figure 6-11. (Note that the program shows only one person's information at a time.)

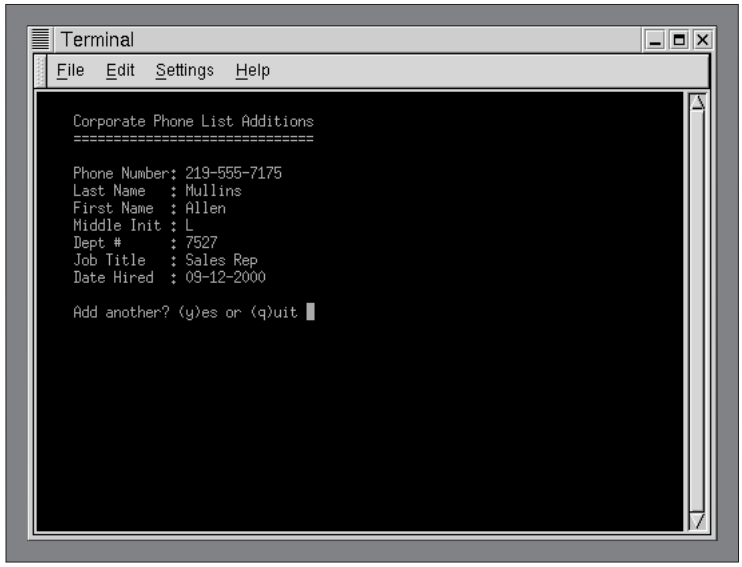

Figure 6-11: Phoneadd script screen

You created the shell script for the corporate phones application. However, you still need to address several deficiencies. For example, how can you return to a previous field as you enter the data? What happens when you enter the same employee twice? What happens if you assign a new employee a phone number that has already been assigned to someone else? Chapter 7, "Advanced Shell Programming," addresses these issues.

SUMMARY

- You can use the tput command to create a simple cursor-placement program to enhance data entry functions.
- You can customize the .bashrc file that resides in the user's home directory to suit the needs of programmers and system administrators.
- You can create aliases and enter them into .bashrc to simplify often-used commands, such as ls –l and rm –i.
- Use the trap command inside a script file to remove temporary files after the script file has been run (exited).
- The grep command serves a key role in the development of shell programs by letting you search and retrieve information from files.
- The Awk program serves as an effective and easy-to-use tool for generating reports.

COMMAND SUMMARY

Chapter 6 Commands

Command	Purpose
alias	Establishes an alias
case…in…esac	A programming structure that allows one action from a set of possible actions to be performed, depending upon the value of a variable
export	Makes a variable an environment variable
for; do…done	Causes a variable to take on each value in a set of values. An action is performed for each value.
if…then…else…fi	Causes one of two actions to be performed depending on a condition
let	Stores arithmetic values in a variable
printenv	Prints a list of environment variables
set –o noclobber	Prevents files from being overwritten by the > operator
tput cup	Moves the screen cursor to a specified row and column
tput clear	Clears the screen
tput cols	Prints the number of columns on the current terminal
tput smso	Enables boldfaced output

Chapter 6 Commands	
tput rmso	Disables boldfaced output
trap	Executes a command when a specified signal is received from the operating system
while; do…done	Repeats an action while a condition exists

(*continued*)

REVIEW QUESTIONS

1. A UNIX alias is _____.
   a. a name that replaces the standard UNIX command name
   b. an alternate name for a UNIX command
   c. always entered into the system-wide initialization file, /etc/profile
   d. preferably entered on the command line each time you log on
2. Use the trap command to _____.
   a. detect errors
   b. monitor for a system signal and take some action
   c. filter out bad data
   d. separate out corrupted information
3. Which of these would place the current directory in the PATH variable?
   a. PATH="$PATH:/~/
   b. PATH="$PATH:."
   c. PATH="$PATH:/home
   d. PATH="$PATH;."
4. You can use the tput command to _____.
   a. add data to the screen
   b. position the cursor on screen
   c. put a character in memory
   d. accept a character from the keyboard
5. You can use the grep command to _____.
   a. search a file for a matching pattern
   b. alter the text in a file
   c. create a mathematical formula
   d. create a pattern of words enclosed with brackets [ ]
6. A script program may access arguments passed to it with _____.
   a. positional parameters, such as $1, $2, etc.
   b. the variables $ARG1, $ARG2, etc.
   c. the argument command
   d. the script_param command

7. Many programmers create a directory in their home directory for storing executable programs. The traditional name for this directory is _____.
   a. programs
   b. executables
   c. prg
   d. bin
8. The file that is executed each time you log on is _____.
   a. .login
   b. autoexec
   c. .bashrc
   d. .welcome
9. The command tput cup is followed on the command line by _____.
   a. the word "clear"
   b. row and column numbers for subsequent screen output
   c. codes to make subsequent screen output old
   d. codes to disable bold output
10. Programmers often set up a directory for holding temporary files. The traditional name for this directory is _____.
    a. temporary
    b. temp
    c. etc
    d. tmp
11. The tput cols command _____.
    a. positions the cursor on the specified column
    b. sets the terminal to the specified number of columns
    c. prints the number of columns available on the current terminal
    d. enables boldfaced printing
12. The tput smso command _____.
    a. enables boldfaced printing
    b. disables boldfaced printing
    c. positions the cursor
    d. clears the screen
13. The tput rmso command _____.
    a. enables boldfaced printing
    b. disables boldfaced printing
    c. positions the cursor
    d. clears the screen

EXERCISES

1. From the command line, issue a command that performs these steps: clears the screen; positions the cursor at row 5, column 30; displays your name; and positions the cursor at row 20, column 0.
2. Create the alias w that executes the command:

   ```
   who –Hi
   ```
3. Store the alias you created in Exercise 2 in the correct file, so it takes effect each time you log on.
4. Change the Phlist1 program to add a heading over the displayed fields when the file is viewed, for example:

   Last Name First Name MI Phone Number Dept# Title Date Hired

   ==========

DISCOVERY EXERCISES

1. Remove the colon characters from the Phoneadd program's screen, and reposition the cursor to improve the appearance of the data entry.
2. Design and write a smaller version of the menu program.
3. Create a small shell program to display the employee phone records showing only their names and the date hired.
4. Create a script file that clears the screen, places two highlighted (boldfaced) lines on row 5, column 10, and row 6, column 12. Display two more, non-boldfaced lines on row 8, column 10, and row 10, column 10.
5. Create a file, music, that contains information on your CD collection. Each record in the file should have these fields:
   Artist Name
   CD Name
   Date Purchased
   Cost
6. Next, create a script file with a menu that lets you view the contents of the file, search for all CDs from a particular artist, and add new records to the file.
7. Design an Awk statement that formats and displays the records in the music file.
8. Describe how you would use the trap command to display the message, "That's all folks," when a script file exits.